Agent-Based Economic Models in OpenCL

Dan Greenwald and Kevin Mullin

December 28, 2012

Introduction

- Traditional macroeconomics assumes perfect insurance and asset markets to make sure that economic activity only depends on aggregate variables.
- However, borrowing constraints, uninsurable income risk, and market incompleteness are important features for understanding economic behavior.
- Relaxing these simplifying assumptions is difficult, because economic activity now depends on the entire distribution of agents' states — requires large scale simulations to solve.
- Solution of optimal policy and simulation can be computationally expensive problems in this type of model, but are well suited to parallelization.
- Our example yields huge performance improvements on the GPU using OpenCL.

Model Environment

- The economy has a macro state z that can correspond to either recession ($z = 0$), or expansion ($z = 1$).
- Infinitely-lived agents (indexed by i) can either be unemployed ($e_i = 0$), or employed ($e_i = 1$).
- Define $s_i = (z, e_i)$, and assume that s_i follows a Markov chain with transition matrix P.
- Agents' income depends on both macro state and employment state, denote by $y(s_i)$.

Borrowing and Saving

- Agents can save and borrow from each other using one-period loans.
- Equivalently: agents hold positive or negative positions in a one-period risk free bond, denoted by b_i.
- You buy (sell) this bond at market price q today, and receive (pay) one unit of consumption next period.
- At equilibrium, q must be set so that the market clears in each period (total saving equals total borrowing).
- Each agent has an identical borrowing limit $b_i \geq -\bar{b}$.

Optimality Conditions

- Need to determine optimal policy of the agent (c_i and b_i) as functions (x_i, q, s_i), where x_i is starting wealth (from previous bonds).
- Optimal policy uniquely determined by

$$qu'(c_i) \geq \beta \mathbb{E}\left[u'(c_i')|s_i\right]$$

  - Must hold with equality for $b_i > -\bar{b}$ (complementary slackness).
  - β is the discount factor (patience).
  - $\mathbb{E}$ is the expectation operator.
  - Primes represent next values (and derivatives — sorry!).

Solution Algorithm

- Want to solve for optimal consumption policy $c(x, q, s)$.
- Since x and q are continuous variables, this is an infinite-dimensional object, so approximate on a set of gridpoints $(\bar{x}_0, \ldots, \bar{x}_{N_x-1})$, $(\bar{q}_1, \ldots, \bar{q}_{N_q-1})$, and use bilinear interpolation between gridpoints.
- Strategy: initialize c^0 with some reasonable starting point (i.e., consume all assets), and iterate on

$$qu'(c^{n+1}(x_i, q, s_i)) \geq \beta \sum_{s_i'} P(s_i, s_i')u'(c^n(x_i', q', s_i')) \qquad (1)$$

until $\max(|c^{n+1} - c^n|) < \varepsilon$.

Endogenous Grid Method

- Simple method: for each current gridpoint (x_i, q, s_i), solve (1) using a nonlinear equation solver.
- This is slow, requires many guesses and function evaluations for each gridpoint.
- Better: start on grid of bond holdings, exploit the fact that $x_i' = b_i + y(s_i')$, and then invert (1) using

$$c_i^* = (u')^{-1}\left\{\beta q^{-1}\sum_{s'} P(s,s')u'(c^n(b_i + y(s_i'), \tilde{q}', s'))\right\}$$

- This defines a (b_i, c_i^*) correspondence, but we can recover starting wealth x_i^* using the budget constraint $x_i^* + y(s_i) = c_i^* + b_i$, to obtain correspondence (x_i^*, c_i^*).
- However, x_i^* does not fall on our original grid, but is an output of the algorithm, so the set of x_i^* is called an *endogenous* grid.

Endogenous Grid Steps

- Use workgroups of size $(K_x, 1, N_s)$, where K_x is some local x size.
- Step 1: calculate $c^n(b_i + y(s_i'), \tilde{q}, s_i')$ using bilinear interpolation (easy because grid is known) and store in local memory.
- Step 2: calculate

  $$\mathbb{E}\left[c^n(b_i + y(s_i'), \tilde{q}, s_i')|s_i\right] = \sum_m P(s_i, s_i')c^n(x_i, \tilde{q}, s')$$

  using previous results.
- Step 3: invert (1) to obtain (c_i^*, x_i^*) correspondence given (q, s_i).

Recovering Original Grid

- No good if each (q, s_i) associated with unique grid, need to get back from endogenous x_i^* grid onto original x_i grid.
- For each x_i gridpoint, search to find corresponding bin on x_i^* grid, and perform linear interpolation to find $c(x_i, q, s_i)$.
- Problem: the relevant x_i^* points are distributed across work groups.
- Solution: load the entire x_i grid one $(K_x \times 1)$ sized block at a time into each work group.
- Each work item takes one x_i from this block, and checks if it falls in that group's x_i^* grid.
- If so, search for exact bin (using bisection) and interpolate.
- Need to overlap the work groups so that every point will fall into one of these intervals.

Data Transfer and Convergence

- To avoid read/write delays, keep arrays for current and previous c array on the device at all times.
- After each iteration, check (on device, in parallel) whether $|c^{n+1} - c^n| \geq \varepsilon$ for each work item. If so, not done!
- No global write issues because we only care if *any* points are too far.
- Only data transfer between host and device on each iteration is flag indicating convergence, arrays written/read only once.

Simulation

- Solution results from the previous section tell the agents what to do given (x_i, q, s_i).
- Apply solution to large-scale simulation (many agents, many periods) to obtain aggregate results.
- Main obstacle: need to solve for the unique q that clears the bond market in *every* period.

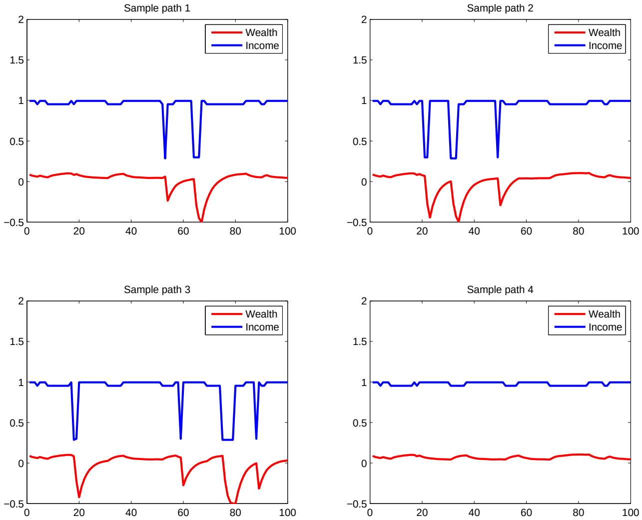
Simulation Algorithm

- Each period, 1-d parallelization using work groups of K_{SIM} agents.
- Given a guess for q, random draws of s_i, and the x_i implied by past behavior, solve for c_i and b_i.
- Add b_i across agents using a reduction algorithm.
    - Step 1: to add all assets in a work group.
    - Step 2: kernel to add assets across work groups.
- Check total assets and adjust q accordingly (bisection).
- Iterate until market clearing, then move to next period.

"Meta" Routine

- All the previous steps assumed a guess for $\tilde{q}(z)$, the average bond price in each macro (z) state.
- Want these expectations to be unbiased.
- Starting with some guess for $\tilde{q}$, run the entire algorithm, and calculate sample means of bond prices in each z state.
- If sample means match $\tilde{q}$ within tolerance, you are done.
- Otherwise, restart the routine using the previous sample means as $\tilde{q}$.

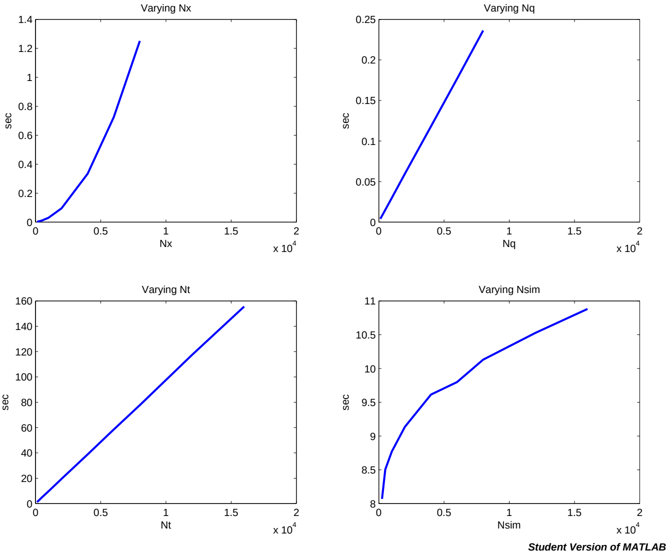
Sample Paths



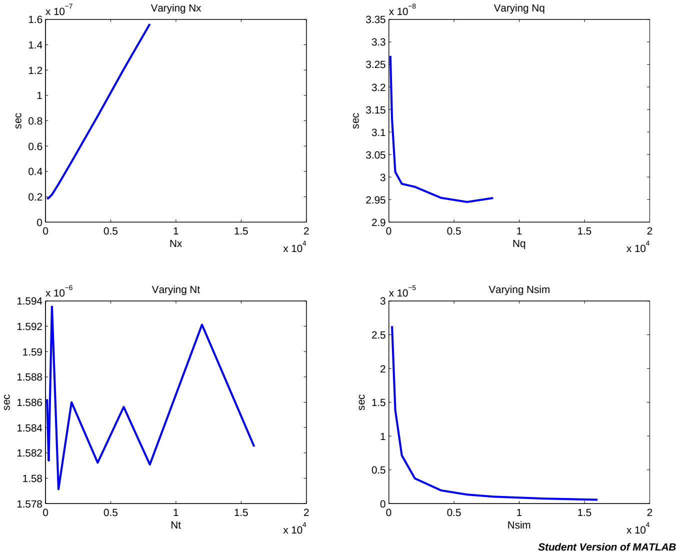
Performance

- Substantial speedup on the GPU, more so for solution algorithm than for simulation algorithm.
- Roughly 1-6x speedup for the simulation algorithm depending on scale.
- Roughly 8-1000x speedup for the solution algorithm depending on scale.
- Diminishing returns to increasing N_x on GPU.
- Roughly linear in N_t, N_q on GPU.
- Timing not sensitive to increases in N_{SIM} on GPU.

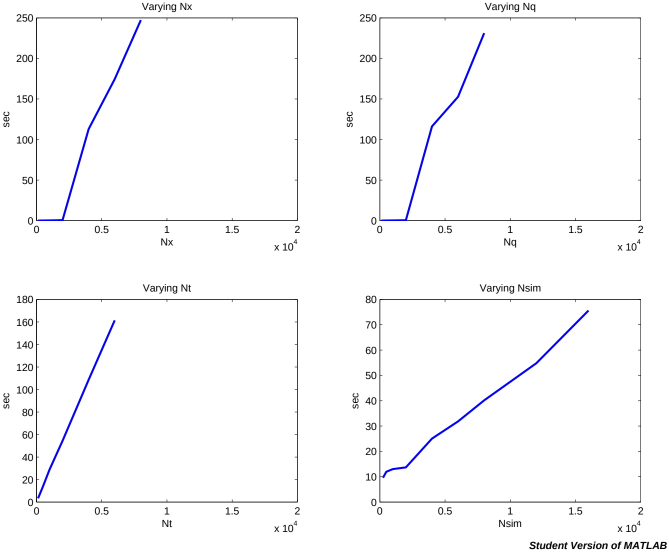
Timings on NVIDIA Tesla M2070 GPU



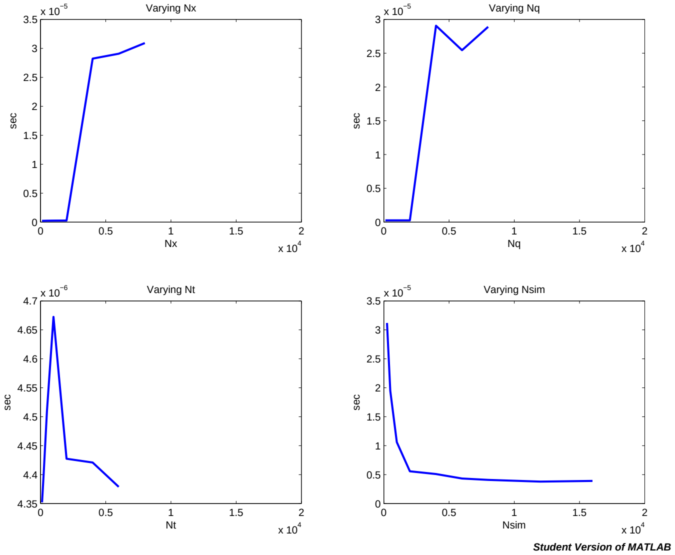
Timings Per Gridpoint on NVIDIA M2070 Tesla GPU



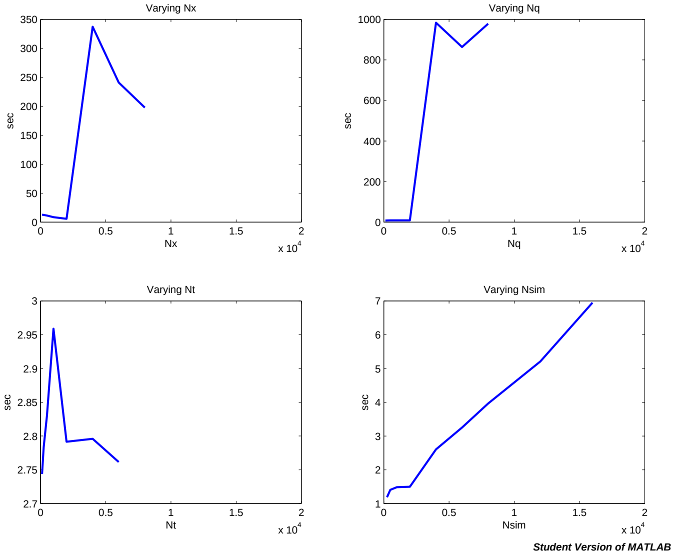
Timings on Intel Xeon CPU



Timings Per Gridpoint on Intel Xeon CPU



Speedup: Ratio of CPU Timing to GPU Timing



Tricks We Learned

- Replace constants with preprocessor macros.
- Similarly, re-use variables using macros, by assigning multiple names to the same object.
- Allocate local memory using clSetKernelArg
- `constant` global memory is good — but there is a limit on how much of this memory you can use on the GPU!